CHAPTER 1

INTRODUCTION

1.1 Background

Information can be said to be the single most important business asset today and achieving a high level of information security can be viewed as imperative in order to maintain a competitive edge. SQL Injection Attacks (SQLIA's) are one of the most severe threats to web application security [29]. They are frequently employed by malicious users for a variety of reasons like financial fraud, theft of confidential data, website defacement, etc. The number of SQLIA's reported in the past few years has been showing a steadily increasing trend and so is the scale of the attacks. It is, therefore, of vital importance to prevent such types of attacks, and SQLIA prevention has become one of the most active topics of research in the industry and academia. There has been significant progress in the field and a number of models have been proposed and developed to counter SQLIA's, but none have been able to guarantee an absolute level of security in web applications, mainly due to the diversity and scope of SQLIA's. One common programming practice in today's times to avoid SQLIA's is to use database stored procedures instead of direct SQL statements to interact with underlying databases in a web application, since these are known to use parameterised queries and hence are not prone to the basic types of SLQIA's. However, there are vulnerabilities in this scheme too, most notably when dynamic SQL statements are used in the stored procedures, to fetch the database objects during runtime.

1.2 Problem

Given the perspective of time passed since web applications entered the commercial market, SQL injection is hardly a new threat [24]. The problem has been described by many security professionals and hackers, and the information is widely spread on the Internet. Many attempts have also been made in order to find countermeasures that can

contend with and overcome SQL injection threats. These countermeasures build on earlier work that covers broader aspects of computer security, including database security issues mentioned above, and the software development process itself.

In addition, the countermeasures constitute new solutions regarding application layer vulnerabilities in general and SQL injection threats in particular. It even seems to be a somewhat common assumption among writers to think that protecting web applications from SQL injection is an easy task, as long as you have an understanding of the SQL injection threat [14]. Nevertheless, corporations, security professionals and hackers continue to announce that SQL injection vulnerabilities are inherent in web applications and reports of compromised applications are frequently published [1, 16, 24]. This clearly indicates that there still seems to exist a lack of awareness, knowledge and respect of SQL injection threats inherent in web applications among security professionals. One reason might be that software development companies and third party vendors do not take a structured security approach when developing web applications. Another reason is that software developers compete in introducing software to the market. We can think of two other reasons as well. First, the vast amount of information published, including detailed step-by-step guides of how to attack web applications with SQL injection, is of course also available for potential attackers. The other reason, we think that the area of SQL injection has never been properly surveyed and that the countermeasures and prevention techniques proposed has not always been systematically composed. We believe that some of the proposed prevention techniques may contain weaknesses and that they therefore can not adequately cope with SQL injection.

1.3 Objective and Outline

Objective of this work is :

1. To show how unauthorized SQL commands can be injected in stored procedure
2. To detect the SQL injection in stored procedure
3. To show how SQL injection can be prevented in stored procedure

1.4 Literature Review

Tim Berners-Lee proposed first concept about the global information network (World Wide Web) in 1990 [2], naming this platform web, this is the initial point of webpage development. Web page is only static picture at first till CGI was released in 1993, then webpage entered dynamic era. In the end, only less people use CGI because of the defect of its security question.

Until the introduction of JSP, PHP and ASP successively, MVC concept was implemented, it allows users not need to install each of huge software anymore, they can utilize the browser to execute the application program. AJAX(Asyn-chronous JavaScript and XML) is a webpage application technology with inter-action structure. This is created by Jesse James Garrett in 2005 [9], using the greatest advantage of Ajax to maintain the data without renewing whole page. This enables Web application program to reflect user's instructions promptly, similar as you double click the program shortcut on your personal computer desktop.

The tool used on Web program safety analysis and inspection (WebSSARI) [11] was launched in May, 2004. It can evaluate the safety level of Web application program automatically and get good result with real tests. Livshits and Lam also proposed one tool LAPSE [13] in Oct.,2005, it can integrate analysis result into Eclipse. It can help find out the vulnerability of incoming unverified data based on static analysis, further it looks for JAVA source code to inform user of all potential weakness. OWASP (Open Web Application Security Project) also issues WebScarab [20] projectAit also integrates the tools to inspect Web security, making the security check efficiently.

CHAPTER 2

WEB APPLICATION

2.1 Introduction

Web application, or webapp, is the general term that is normally used to refer to all distributed web-based applications. According to the more technical software engineering definition, a web application is described as an application accessible by the web through a network. Many companies are converting their computer programs into web-based applications. Web Applications are similar to computer-based programs but differ only in that they are accessible through the web, allowing the creation of dynamic websites and providing complete interaction with the end-user. Web Applications are placed on the Internet and all processing is done on the server, the computer which hosts the application.

Web applications are sets of web pages, files and programs that reside on a Company's web server, which any authorized user can access over a network such as the World Wide Web or a local intranet. A web application is usually a three-tiered model. Normally, the first tier is a Web browser on the client side, the second is the real engine on the server-side where the applications core runs, and the third layer is a database. A server processes all user transactions and usually the end-user simply accesses the web application by a Web browser, interacting with it. Since web applications reside on a server, they are easy to manage. In fact, they can be updated and modified at any time by the web applications owner with minimal effort and without any distribution or installation of software on the client's machines. This is the main reason for the widespread adoption of Web applications in today's organizations [29]

Nowadays, web applications are becoming increasingly popular and are poised to become a major player in the overall software market due to the benefits they afford, such as visibility and worldwide access. They are, without a doubt, essential to the current and next generation of businesses and they have become part of our everyday online lives. In

fact, a web application is a worldwide gate accessible not only through standard personal computers but also though different communication devices such as mobile phones and PDAs. The use of web applications is especially beneficial for a company: with just a little investment, a company can open up a marketing channel that will allow potential clients easy global access to its business 24 hours a day.

2.2 Architecture of Web application

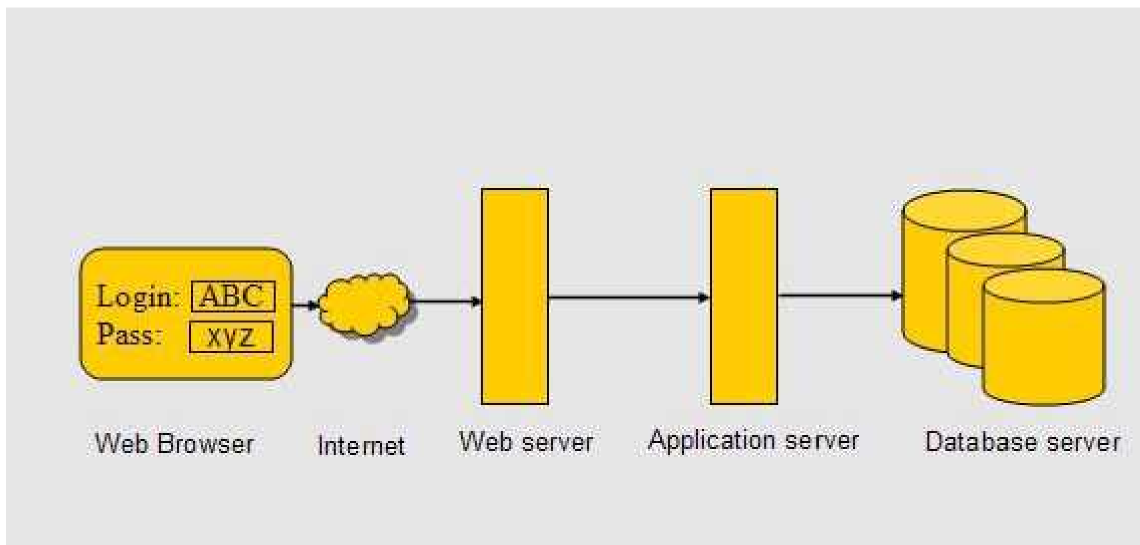


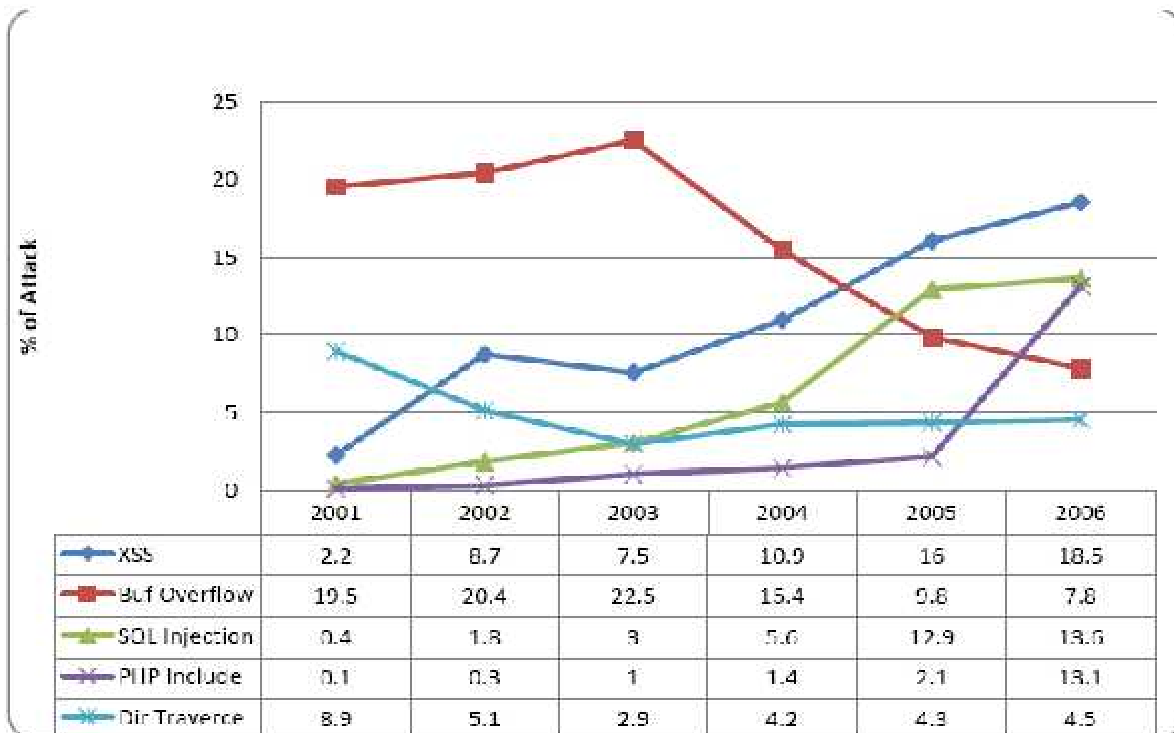
Figure 1: *Architecture of a typical web based system*

A web system consists of a web browser at the user end. The user is connected to the web application through the internet. A firewall protects the web system from intrusion and allows traffic at port 80 only. The web server receives request from the browser, processes them and passes the dynamic part to the application server, which processes server side code like JSP. All requests for database access are passed to the database server. The results are than shipped back to the web browser as HTML web pages.

2.3 Input Validation-Based Vulnerabilities

The most prominent class of input validation errors are SQL injections. SQL injections are the classes of vulnerabilities in which an attacker causes the web application server to

produce HTML documents and database queries, respectively, that the application programmer did not intend. They are possible because, in view of the low-level APIs described above for communication with the browser and database, the application constructs queries and HTML documents via low-level string manipulation and treats un-trusted user inputs as isolated lexical entities. This is especially common in scripting languages such as PHP, which generally do not provide more sophisticated APIs and use strings as the default representation for data and code. Some paths in the application code may incorporate user input unmodified or unchecked into database queries or HTML documents. The modifications/checks of user input on other paths may not adequately constraint the input to function in the generated query or HTML document as the application programmer intended. In that sense, SQL injections are integrity violations in which low-integrity data is used in a high-integrity channel; that is, the browser or the database executes code from an un-trusted user, but does so with the permissions of the application server



	2001	2002	2003	2004	2005	2006
XSS	2.2	8.7	7.5	10.9	16	18.5
Buf Overflow	19.5	20.4	22.5	15.4	9.8	7.8
SQL Injection	0.4	1.8	3	5.6	12.9	13.6
PHP Include	0.1	0.3	1	1.4	2.1	13.1
Dir Traverse	8.9	5.1	2.9	4.2	4.3	4.5

***Figure 2**: Vulnerability Type*

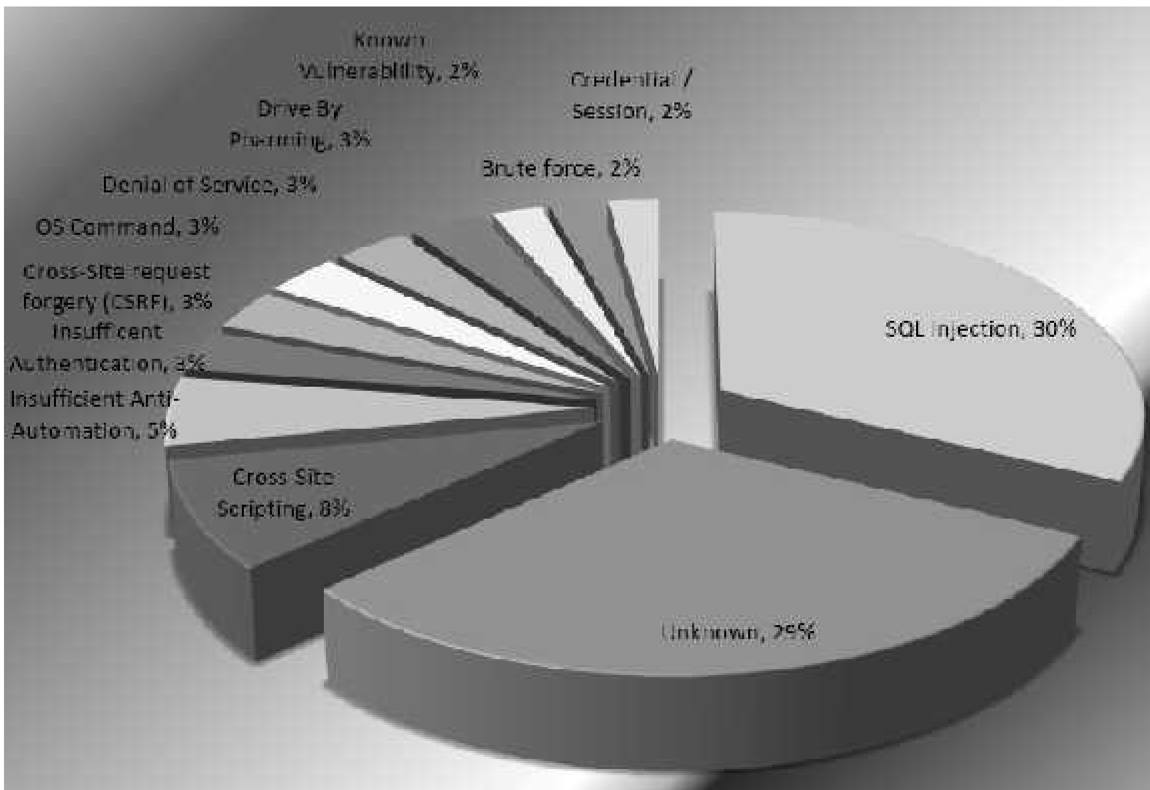


Figure 3: *Reported Web Attacks in 2008*

To highlight how ubiquitous web applications have become and how prevalent their problems are, Figure-2. shows, for each year from 2001 to 2006, the percentage of newly reported security vulnerabilities in five vulnerability classes (this data comes from Mitre's report [22]): XSS, SQL injection, PHP file inclusions, buffer overflows, and directory traversals. These were the five most reported vulnerabilities in 2006. All of these except buffer overflows are specific to web applications. Note that SQL injections are consistently at or near the top: 13% of the reported vulnerabilities in 2006, respectively. Some web security analysts speculate that because web applications are highly accessible and databases often hold valuable information, the percentage of SQL injection attacks being executed is significantly higher than the percentage of reported vulnerabilities would suggest. Empirical data supports this hypothesis. Figure-3 shows percentages of reported web attacks for the year 2008 (this data comes from the Web Hacking Incidents Database) [3]. Although many attacks go unreported or even undetected, this chart shows that 30% of the web-based attacks that made the press in 2008 were SQL injections, respectively.

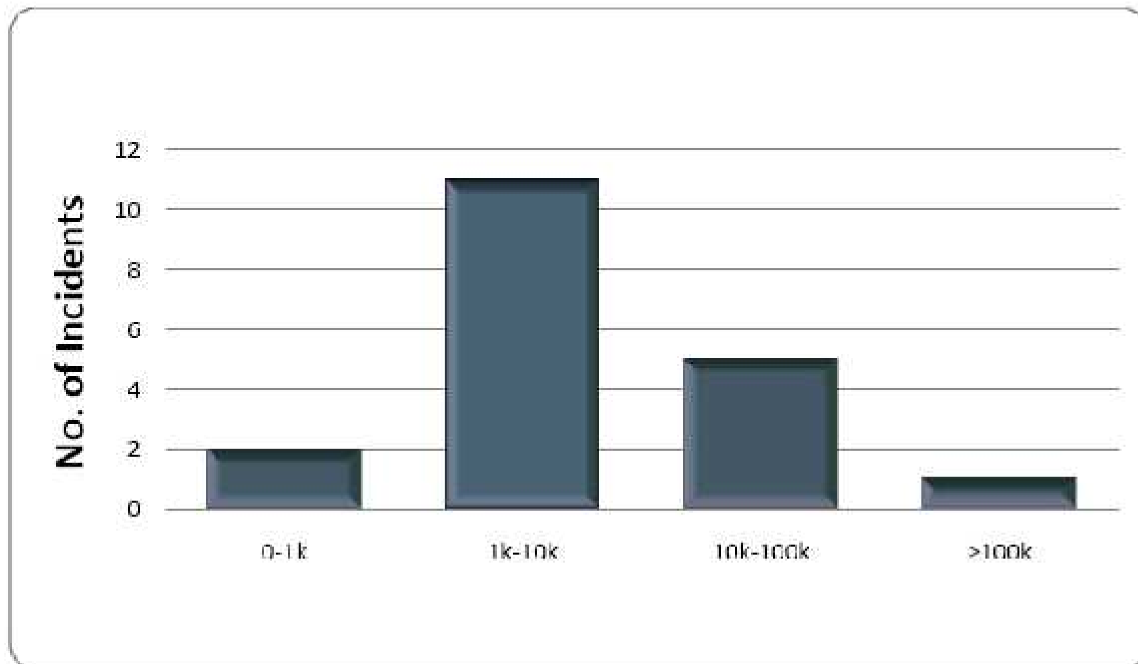


Figure 4: *Number of incident by number of records leaking in 2007*

This kind of attacks can cause severe damage. Typical uses of SQL injection leak confidential information from a database, by-pass authentication logic, or add unauthorized accounts to a database. Figure-4 shows the number of reported information leakage attacks in 2007 that leaked the number of records in each of four ranges split by a log-scale(this data comes from the Web Hacking Incidents Database [19]). A news article from May, 2008 gives an example of the kinds of records that Over 1.5 million pages affected by via an SQL injection. It is an attempt to mitigate the impact of the recent waves of SQL injection attacks, and provide more transparency into the approximate number of affected pages [7].

2.4 Web Application and SQL Injection

Most of the current web applications use RDBMS (Relational Database Management Systems). Sensitive information like credit card, social security and financial records are stored in these databases. Web application allows users to view, edit or store information in RDBMS through programs written by web application programmers. Usually

programmers who write these programs are unaware of technique for writing secure code. They would focus on implementing desired functionalities and would focus less on security aspects. This results in vulnerabilities in web applications. Vulnerabilities allow attacker to target these web application and obtain valuable information. Attackers would send SQL (Structured Query language) to interact with RDBMS servers or modify existing SQL to retrieve unauthorized information without any authentication. The risk is higher if the application is open source or if the attacker is able to gain source code through other means. In this case he can analyze the code to find out the vulnerabilities

CHAPTER 3

SQL

3.1 Introduction

The SQL standard, according to Connolly et al. [25], was defined by the American National Standards Institute (ANSI) and was later adopted by the International Standards Organization (ISO). Its objectives are to allow users to create database and relation structures, managing tables by inserting, modifying, and deleting data as well as retrieve information from the database through queries. SQL queries are commands that are passed to the RDBMS, and specify which data is to be gathered from one or more tables and how it should be arranged. We intend to follow the ISO SQL standard used by Connolly et al., and will be using it throughout this thesis unless stated otherwise.

SQL consists of two major components: the Data Manipulation Language (DML) and the Data Definition Language (DDL). Using the DML, users can manipulate data stored inside tables in the database, while the DDL allows creating and destroying database objects such as schemas, domains, tables, views and indices.

3.1.1 DML

The Data Manipulation Language has four available statements, namely SELECT, INSERT, UPDATE and DELETE.

SELECT used for retrieving information from one or more tables in the database and displaying it.
INSERT used for adding new data rows in a table.
UPDATE used for modifying data rows in a table.
DELETE used for removing data rows from a table.

3.1.2 DDL

Connolly et al. [25] defines the Data Definition Language (DDL) as ”A descriptive language that allows the DBA or user to describe and name the entities required for the application and the relationships that may exist between the different entities.” Thus, the DDL is used when manipulating the database’s meta-data, which describes the objects contained in the database and allows access to them. The DDL does not allow users to manipulate data stored in the database.

3.2 Query Techniques

An SQL query to be executed in a RDBMS can be constructed using two techniques. Either the query is allowed to be dynamically tailored with respect of both SQL keywords and query arguments, or the query syntax is unchangeable, only allowing arguments to be passed [25].

3.2.1 Dynamic SQL

Dynamic SQL refers to the concept of allowing an SQL query to be dynamically built by concatenating statements and using variables that supply the query with dynamic values. According to Connolly et al. [25] and Harper [18] and Khatri [10], the query is typically stored in a variable and the query builders consist of application logic components that adds SQL syntax and arguments to the variable in a process governed by specified conditions. Such queries are interpreted and compiled at run-time by the RDBMS, meaning that the query will be compiled every time it is executed. Since dynamic SQL allows SQL syntax to be added, both SQL keywords and values may be passed as arguments to queries.

3.2.2 Static SQL

Static SQL refers to the concept of using fixed and unchangeable SQL queries. Such queries are predefined and compiled and are not permitted to add SQL keywords, defined in DDL or DML. Only arguments to clauses, e.g. WHERE, may be allowed to be passed to the queries. Either the query is embedded in application logic code in form of prepared statements or it resides in RDBMS as stored procedures.

CHAPTER 4

SQL INJECTION

4.1 SQL Injection Attacks

SQL injection is the act of passing SQL code into an application that was not intended by the developer. SQL injection vulnerability can occur when a program uses user-provided data in a database query without proper input validation. On the other hand SQL injection is a form of attack on a database-driven web site in which the attacker executes unauthorized SQL commands by taking advantage of insecure code on a system connected to the Internet.

SQL injection is a particularly dangerous threat that exploits application layer vulnerabilities inherent in web applications. Instead of attacking instances such as web servers or operating systems, the purpose of SQL injection is to attack RDBMSs, running as back-end systems to web servers, through web applications. [15]

More specifically, attackers can bypass existing security mechanisms implemented to enforce security services, and may therefore gain access to and manipulate information assets outside their privileges. This is accomplished by modifying input parameters expected in fields of forms embedded in web pages, in order to change the underlying queries built with SQL and passed to the database through the web server. Another method is to insert arbitrary SQL directly in the query string portion of an URL in the address field of web browsers. [5, 6, 15]

Every web application, using a relational database, can theoretically be a subject for SQL injection attacks. Those databases usually contain corporation's most valuable information assets: corporate and customer data. Those data are vital for the functions of a corporation's web applications, but often even more crucial and valuable for the corporation itself: user credentials, sensitive financial information, preferences, invoices, payments, inventory data etc. If successful, SQL injection attacks may therefore result in

exposure of and serious impact on the corporations most valuable information assets. These attacks may in the worst case result in a completely destroyed database schema, which in turn may affect a corporation's ability to perform business. [15, 18]

The typical intentions of the attacker performing a SQL injection attacks may be to:

- Identify inject-able parameters.
- Perform database finger-printing.
- Determine the database schema.
- Extract and modify data.
- Perform Denial of Service (DoS)
- Bypass authentication and perform privilege escalation
- Execute remote commands

4.2 Attack Intention

When a threat agent utilizes a crafted malicious SQL input to launch an attack, the attack intention is the goal that the threat agent tries to achieve once the attack has been successfully executed.

Identifying Injectable Parameters [26]: Injectable parameters are the parameters or the user input fields of the Web applications directly used by server-side program logic to construct SQL statements, which are vulnerable to SQLIA. In order to launch a successful attack, a threat agent must first discover which parameters are vulnerable to SQL injection attack.

Performing database finger-printing [26]: Database finger-print is the information that identifies a specific type and version of database system. Every database system employs a different proprietary SQL language. For example, the SQL language employed by Microsoft SQL server is T-SQL while Oracle SQL server uses PL/SQL. In order for an attack to be succeeded, the attacker must first find out the type of and version of database deployed by a web application, and then craft malicious SQL input accordingly.

Determining database schema [26]: Database schema is the structure of the database system. The schema defines the tables, the fields in each table, and the relationships between fields and tables. Database schema is used by threat agents to compose a correct subsequent attack in order to extract or modify data from database.

Bypassing Authentication [26]: Authentication is a mechanism employed by web application to assert whether a user is who he/she claimed to be. Matching a user name and a password stored in the database is the most common authentication mechanism for web applications. Bypassing authentication enables an attacker to impersonate another application user to gain un-authorized access

Extracting Data [26]: In most of the cases, data used by web applications are highly sensitive and desirable to threat agents. Attacks with intention of extracting data are the most common type of SQL injection attacks

Adding or Modifying Data [26]: Database modification provides a variety of gains for a threat agent, for instance, a hacker can pay much less for a online purchase by altering the price of a product in the database. Or, the threads in a online discussion forum can be modified by an attacker to launch subsequent Cross-Site-Scripting attacks.

Performing denial of service [26]: These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category.

Executing Remote Commands [26]: Remote commands are executable code resident on the compromised database server. Remote command execution allows an attacker to run arbitrary programs on the server. Attacks with this type of intention could cause entire internal networks being compromised.

Performing privilege escalation [26]: Privileges are described in a set of rights or permissions associated with users. Privilege escalation allows an attacker to gain un-

authorized access to a particular asset by associating a higher privilege set of rights with a current user or impersonate a user who has higher privilege

Downloading File: Downloading files from a compromised database server enable an attacker to view file content stored on the server. If the target web application resides on the same host, sensitive data such as configuration information and source code will be disclosed too.

Uploading File: Uploading files to a compromised database server enable an attacker to store any malicious code onto the server. The malicious code could be a Trojan, a back door or a worm that can be used by an attacker to launch subsequence attack.

4.3 How it Happens

SQL is the standard language for accessing Microsoft SQL Server, Oracle, MySQL, sybase, and Informix (as well as other) database servers. Most Web applications need to interact with a database, and most Web application programming languages, such as ASP, C#, .NET, Java, and PHP, provide programmatic ways of connecting to a database and interacting with it. SQL injection vulnerabilities most commonly occur when the Web application developer does not ensure that values received from a Web form, cookie, input parameter, and so forth are validated before passing them to SQL queries that will be executed on a database server. If an attacker can control the input that is sent to an SQL query and manipulate that input so that the data is interpreted as code instead of as data, the attacker may be able to execute code on the back-end database.

Each programming language offers a number of different ways to construct and execute SQL statements, and developers often use a combination of these methods to achieve different goals. A lot of Web sites that offer tutorials and code examples to help application developers solve common coding problems often teach insecure coding practices and their example code is also often vulnerable. Without a sound understanding of the underlying database that they are interacting with or a thorough understanding and

awareness of the potential security issues of the code that is being developed, application developers can often produce inherently insecure applications that are vulnerable to SQL injection.

4.3.1 Dynamic String Building

Dynamic string building is a programming technique that enables developers to build SQL statements dynamically at runtime. Developers can create general-purpose, flexible applications by using dynamic SQL. A dynamic SQL statement is constructed at execution time, for which different conditions generate different SQL statements. It can be useful to developers to construct these statements dynamically when they need to decide at runtime what fields to bring back from, say, SELECT statements, the different criteria for queries, and perhaps different tables to query based on different conditions.

However, developers can achieve the same result in a much more secure fashion if they use parameterized queries. Parameterized queries are queries that have one or more embedded parameters in the SQL statement. Parameters can be passed to these queries at runtime; parameters containing embedded user input would not be interpreted as commands to execute, and there would be no opportunity for code to be injected. This method of embedding parameters into SQL is more efficient and a lot more secure than dynamically building and executing SQL statements using string-building techniques.

The following PHP code shows how some developers build SQL string statements dynamically from user input. The statement selects a data record from a table in a database. The record that is returned depends on the value that the user is entering being present in at least one of the records in the database.

```
// a dynamically built sql string statement in PHP
$query = "SELECT * FROM table WHERE field = '$_GET["input"]'";
```

One of the issues with building dynamic SQL statements such as this is that if the code does not validate or encode the input before passing it to the dynamically created statement, an attacker could enter SQL statements as input to the application and have his

SQL statements passed to the database and executed. Here is the SQL statement that this code builds:

*SELECT * FROM TABLE WHERE FIELD = 'input'*

Incorrectly Handled Escape Characters

SQL databases interpret the quote character (‘) as the boundary between code and data. It assumes that anything following a quote is code that it needs to run and anything encapsulated by a quote is data. Therefore, you can quickly tell whether a Web site is vulnerable to SQL injection by simply typing a single quote in the URL or within a field in the Web page or application. Here is the source code for a very simple application that passes user input directly to a dynamically created SQL statement:

```
// build dynamic SQL statement
$SQL = "SELECT * FROM table WHERE field = '$_GET["input"]'";
// execute sql statement
$result = mysql_query($SQL);
// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);
// iterate through the record set returned
$row = 1;
while ($db_field = mysql_fetch_assoc($result)) {
if ($row <= $rowcount){
print $db_field[$row] . "<BR>";
$row++;
}
```

To enter the single-quote character as input to the application, may be presented with either one of the following errors; the result depends on a number of environmental factors, such as programming language and database in use, as well as protection and defense technologies implemented:

Warning: mysql_fetch_assoc(): supplied argument is not a valid MySQL result Resource

You may receive the preceding error or the one that follows. The following error provides useful information on how the SQL statement is being formulated:

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ''VALUE'''

The reason for the error is that the single-quote character has been interpreted as a string delimiter. Syntactically, the SQL query executed at runtime is incorrect (it has one too many string delimiters), and therefore the database throws an exception. The SQL database sees the single-quote character as a special character (a string delimiter). The character is used in SQL injection attacks to "escape" the developer's query so that the attacker can then construct his own queries and have them executed.
The single-quote character is not the only character that acts as an escape character; for instance, in Oracle, the blank space ( ), double pipe (||), comma (,), period (.), (*/), and double-quote characters (") have special meanings.

Incorrectly Handled Types

The single-quote character is interpreted as a string delimiter and is used as the boundary between code and data. When dealing with numeric data, it is not necessary to encapsulate the data within quotes; otherwise, the numeric data would be treated as a string.

Here is the source code for a very simple application that passes user input directly to a dynamically created SQL statement. The script accepts a numeric parameter ($userid) and displays information about that user. The query assumes that the parameter will be an integer and so is written without quotes.

```
// build dynamic SQL statement
$SQL = "SELECT * FROM table WHERE field = $_GET["userid"]"
// execute sql statement
$result = mysql_query($SQL);
// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);
// iterate through the record set returned
$row = 1;
while ($db_field = mysql_fetch_assoc($result)) {
if ($row <= $rowcount){
print $db_field[$row] . "<BR>";
$row++;
}
}
```

MySQL provides a function called LOAD_FILE that reads a file and returns the file contents as a string. To use this function, the file must be located on the database server host and the full pathname to the file must be provided as input to the function. The calling user must also have the FILE privilege. The following statement, if entered as input, may allow an attacker to read the contents of the /etc/passwd file, which contains user attributes and usernames for system users:

```
1 UNION ALL SELECT LOAD_FILE('/etc/passwd')--
```

The attacker's input is directly interpreted as SQL syntax; so, there is no need for the attacker to escape the query with the single-quote character. Here is a clearer depiction of the SQL statement that is built:

```
SELECT * FROM TABLE
WHERE
USERID = 1 UNION ALL SELECT LOAD_FILE('/etc/passwd')—
```

Incorrectly Handled Query Assembly

Some complex applications need to be coded with dynamic SQL statements, as the table or field that needs to be queried may not be known at the development stage of the application or it may not yet exist. An example is an application that interacts with a large database that stores data in tables that are created periodically. A fictitious example may be an application that returns data for an employee's time sheet. Each employee's time sheet data is entered into a new table in a format that contains that month's data (for January 2008 this would be in the format employee_employee-id_01012008). The Web developer needs to allow the statement to be dynamically created based on the date that the query is executed.

The following source code for a very simple application that passes user input directly to a dynamically created SQL statement demonstrates this. The script uses applicationgenerated values as input; that input is a table name and three column names. It then displays information about an employee. The application allows the user to select what data he wishes to return; for example, he can choose an employee for which he would like to view data such as job details, day rate, or utilization figures for the current month. Because the application already generated the input, the developer trusts the data; however, it is still user-controlled, as it is submitted via a GET request. An attacker could submit his table and field data for the application-generated values.

```
// build dynamic SQL statement
$SQL = "SELECT $_GET["column1"], $_GET["column2"], $_GET["column3"] FROM
$_GET["table"]";
// execute sql statement
$result = mysql_query($SQL);
// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);
// iterate through the record set returned
$row = 1;
```

```
while ($db_field = mysql_fetch_assoc($result)) {
if ($row <= $rowcount){
print $db_field[$row] . "<BR>";
$row++;
}
}
```

If an attacker was to manipulate the HTTP request and substitute the users value for the table name and the user, password, and Super_priv fields for the application-generated column names, he may be able to display the usernames and passwords for the database users on the system. Here is the URL that is built when using the application:
http://www.victim.com/user_details.php?table=users&column1=user&column2=password&column3=Super_priv

If the injection were successful, the following data would be returned instead of the time sheet data. This is a very contrived example; however, real-world applications have been built this way.

user	Password	Super_priv
root	2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19	Y
sqlinjection	2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19	N
sanu	2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19	N

***Table 1:**. Data Returned After Successful Injection*

Incorrectly Handled Multiple Submissions

Application developers also tend to design an application around a user and attempt to guide the user through an expected process flow, thinking that the user will follow the logical steps they have laid out. For instance, they expect that if a user has reached the third form in a series of forms, the user must have completed the first and second forms. In reality, though, it is often very simple to bypass the expected data flow by requesting

resources out of order directly via their URLs. Take, for example, the following simple application:

```
// process form 1
if ($_GET["form"] = "form1"){
// is the parameter a string?
if (is_string($_GET["param"])) {
// get the length of the string and check if it is within the
// set boundary?
if (strlen($_GET["param"]) < $max){
// pass the string to an external validator
$bool = validate(input_string, $_GET["param"]);
if ($bool = true) {
// continue processing
            }
           }
     }
}
```

// process form 2

```
if ($_GET["form"] = "form2"){
// no need to validate param as form1 would have validated it for us
$SQL = "SELECT * FROM TABLE WHERE ID = $_GET["param"]";
// execute sql statement
$result = mysql_query($SQL);
// check to see how many rows were returned from the database
$rowcount = mysql_num_rows($result);
$row = 1;
// iterate through the record set returned
while ($db_field = mysql_fetch_assoc($result)) {
if ($row <= $rowcount){
```

```
print $db_field[$row] . "<BR>";
$row++;
}
}
}
```

The application developer does not think that the second form needs to validate input, as the first form will have performed the input validation. An attacker could call the second form directly, without using the first form, or he could simply submit valid data as input into the first form and then manipulate the data as it is submitted to the second form. The first URL shown here would fail as the input is validated; the second URL would result in a successful SQL injection attack, as the input is not validated:

```
[1] http://www.victim.com/form.php?form=form1&param=' SQL Failed --
[2] http://www.victim.com/form.php?form=form2&param=' SQL Success --
```

4.3.2 Insecure Database Configuration

You can mitigate the access that can be leveraged, the amount of data that can be stolen or manipulated, the level of access to interconnected systems, and the damage that can be caused by an SQL injection attack, in a number of ways. Securing the application code is the first place to start; however, you should not overlook the database itself. Databases come with a number of default users preinstalled. Microsoft SQL Server uses the infamous “sa” database system administrator account, MySQL uses the “root” and “anonymous” user accounts, and with Oracle, the accounts SYS, SYSTEM, DBSNMP, and OUTLN are often created by default when a database is created. These aren’t the only accounts, just some of the betterknown ones; there are a lot more! These accounts are also preconfigured with default and well-known passwords.

Some system and database administrators install database servers to execute as the root, SYSTEM, or Administrator privileged system user account. Server services, especially database servers, should always be run as an unprivileged user (in a chroot environment,

if possible) to reduce potential damage to the operating system and other processes in the event of a successful attack against the database. However, this is not possible for Oracle on Windows, as it must run with SYSTEM privileges.

Each type of database server also imposes its own access control model assigning various privileges to user accounts that prohibit, deny, grant, or enable access to data and/or the execution of built-in stored procedures, functionality, or features. Each type of database server also enables, by default, functionality that is often surplus to requirements and can be leveraged by an attacker (xp_cmdshell, OPENROWSET, LOAD_FILE, ActiveX, and Java support, etc.).

Application developers often code their applications to connect to a database using one of the built-in privileged accounts instead of creating specific user accounts for their applications needs. These powerful accounts can perform a myriad of actions on the database that are extraneous to an application's requirement. When an attacker exploits an SQL injection vulnerability in an application that connects to the database with a privileged account, he can execute code on the database with the privileges of that account. Web application developers should work with database administrators to operate a least-privilege model for the application's database access and to separate privileged roles as appropriate for the functional requirements of the application.
In an ideal world, applications should also use different database users to perform SELECT, UPDATE, INSERT, and similar commands. In the event of an attacker injecting code into a vulnerable statement, the privileges afforded would be minimized. Most applications do not separate privileges, so an attacker usually has access to all data in the database and has SELECT, INSERT, UPDATE, DELETE, EXECUTE, and similar privileges. These excessive privileges can often allow an attacker to jump between databases and access data outside the application's data store.

To do this, though, attacker needs to know what else is available, what other databases are installed, what other tables are there, and what fields look interesting! When an attacker exploits an SQL injection vulnerability attacker will often attempt to access

database metadata. Metadata is data about the data contained in a database, such as the name of a database or table, the data type of a column, or access privileges. Other terms that sometimes are used for this information are data dictionary and system catalog. For MySQL Servers (Version 5.0 or later) this data is held in the INFORMATION_SCHEMA virtual database and can be accessed by the SHOW DATABASES and SHOW TABLES commands. Each MySQL user has the right to access tables within this database, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges. Microsoft SQL Server has a similar concept and the metadata can be accessed via the INFORMATION_SCHEMA or with system tables (sysobjects, sysindexkeys, sysindexes, syscolumns, systypes, etc.), and/or with system stored procedures; SQL Server 2005 introduced some catalog views called "sys.*" and restricts access to objects for which the user has the proper access privileges.

Each Microsoft SQL Server user has the right to access tables within this database and can see all of the rows in the tables regardless of whether he has the proper access privileges to the tables or the data that is referenced.

Meanwhile, Oracle provides a number of global built-in views for accessing Oracle metadata (ALL_TABLES, ALL_TAB_COLUMNS, etc.). These views list attributes and objects that are accessible to the current user. In addition, equivalent views that are prefixed with USER_ show only the objects owned by the current user (i.e., a more restricted view of metadata), and views that are prefixed with DBA_ show all objects in the database (i.e., an unrestricted global view of metadata for the database instance). The DBA_ metadata functions require database administrator (DBA) privileges. Here is an example of these statements:

-- Oracle statement to enumerate all accessible tables for the current user
SELECT OWNER, TABLE_NAME FROM ALL_TABLES ORDER BY TABLE_NAME;
-- MySQL statement to enumerate all accessible tables and databases for the
-- current user

```
SELECT table_schema, table_name FROM information_schema.tables;
-- MS SQL statement to enumerate all accessible tables using the system
-- tables
SELECT name FROM sysobjects WHERE xtype = 'U';
-- MS SQL statement to enumerate all accessible tables using the catalog
-- views
SELECT name FROM sys.tables;
```

4.4 Existing Technologies to Stop SQL Injection

4.4.1 Defensive Programming

Defensive Programming is a Programming practice that was done on the integrated application code when the software is in development stage. The programmer tries to minimize all the bugs in the programming, and the programmer tries to find out the way to use the code for hacking purpose. So by this type of coding practice the programmer will be able to find out the security weakness in the code. By securing the code the programmer can possibly stop potential attacks on the website. The code can be analyzed in many ways like reducing the complexity of the program. Doing reviews on the code again and again to find out the possible vulnerabilities of the code and to perform software testing on the code. By forming the programming with the above measures the programmer can develop a code which may be immune. Ultimately the attacker tries to find out new ways to penetrate into the code. When the hacker finds a new method that is not tested while application programming the attacker may be successful at some stage. The user also limited to test the application with the attacking techniques he knows about.

There are a number of ways a programmer/system administrator can prevent or counter attacks made on their systems.

Parameterized Query: Parameterized query is parameterized database access API provided by development platform such as PrepareStatement in Java or SQLParameter

.NET. Instead of composing SQL by concatenating string, each parameter in a SQL query is declared using place holder and input is provided separately.

Least Privilege: The account that an application uses to access the database should have only the minimum permissions necessary to access the objects that it needs to use.

Different Accounts: Use a different database account for a task that requires a different level of privilege.

Customized Error Message: Attacker may gain access to knowledge through overly informative error messages, yet completely removing error messages makes debugging a difficult task. Customized error messages hinder the reconnaissance progress of attacker, particularly in deducing specific details such as inject-able parameters, etc.

Figure 5: *Informative error messages*

System Stored Procedure Reduction: Once a attacker gains knowledge of which back-end server is used, he/she has knowledge of an entire set of system stored procedures that are available. By limiting the system stored procedures one can execute on a server, especially the processes that are not used, one can reduce or even eliminate vulnerabilities that may arise from these stored procedures

SQL Keyword Escaping: Escape specific SQL keyword or delimiter in the input string.

Input Variable Length Checking: By checking for input variable length, malicious code strings beyond certain length limits will not be applicable. Even if the length limitation is long enough to fit a few additional queries, the inability to input an infinitely

long string disables the attacker from employing evasion techniques such as encoding, and consequently, allows signature based detection mechanisms to intercept simple attacks.

Although these techniques remain the best way to prevent SQL injection vulnerabilities, but their application is problematic in practice. These techniques are prone to human error and are not as rigorously and completely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to apply defensive coding practices rigorously and correctly to all sources of input. In fact, many of the SQL injection vulnerabilities discovered in real applications are due to human errors: developers forgot to add checks or did not perform adequate input validation

4.4.2 Anomaly Detection

Anomaly detection technique is a method where the administrator observes the network traffic. By observing the network traffic the administrator can find when there is a possible attack performed against the server. The anomaly system verifies the traffic which is going through the network by analyzing the recorded behavior with the network traffic there is possibility to find out the attacks. The anomaly detection is classified in many types like rule-based, model-based and statistical analysis. The programmer creates a set of rules to define possible types of attacks that can be performed on the program when the rule is not satisfied there might be a possible attack on the database. In the model based approach the application imports the anomaly techniques that are characterized to define attacks on the server. If the incoming traffic doesn‟t meets the model, the application indicates there is a possible attack on the database. Statistical analysis is a different approach where the program calculates the system behavior by measuring certain variables overtime and it takes average point of the calculated variables. If the new traffic exceeds the thresholds, indicates there might be a possible attack going on the server. The anomaly detection techniques are very good in detecting the attacks like Buffer overflows and different kind of attacks but as the user can be able to pass the data in a method which represents like normal traffic these techniques are unsuccessful for detection of SQL Injection attacks.

CHAPTER 5

EXISTING PREVENTION/DETECTION MODELS

5.1 Secure SQL Processing

This model has been proposed by Dibyendu Aich, an M-tech research scholar at the National Institute of Rourkela in the research paper titled “Secure Query processing by blocking sql injection” [30]. The basic mechanism that this model uses is a two phase query analysis, consisting of the static analysis phase and the dynamic analysis phase. During runtime, the model checks the input query structure with the previously stored query structure to determine possible SQLIA’s. The database of the valid query structures is made statically, during compilation. The valid structures are stored as a singly linked list of the different tokens in a sequential ordering. All such valid query structures in the application are then stored as a doubly linked list where each node of the doubly linked list contains the staring address of an individual singly linked list of a valid query structure. So basically, when a new query is sent to the database server, the model starts searching for a match of the structure of the query in the linked representation. If a match is found, the search is stopped and the query is dubbed a valid query, else it is labelled as an SQL injection attack.

In effect, the searching procedure is the operation of checking if the sequence of query language tokens generated by the arrived query is the same as the sequence of tokens generated by at least one singly linked list of valid query structures, upon finding which, the query is sent to the server for execution. This model makes use of the SQL parser of the backend database to parse the incoming query into a sequence of tokens, with an additional field to denote if the node is a user input or if it is a token of the static part of the query. When a node denoting a user input is found in the linked list, the checker skips right past it to the next static token, and the matching continues.

Link to previous node	Link to the singly linked list storing individual query structure	Hit Count	Link to next node

Figure 6: *Node structure of the main doubly linked list*

Data related to the tokens of a valid query	Is it a user input?	Link to next node

Figure 7: *Node structure of singly linked list for storing a valid individual query structure*

From the above description of the matching technique, it is clear that for a successful search, number of tokens in the input query must be equal to the length of the linked list storing its structure.

Key advantages over other token matching algorithms :

- Although this process is a relatively secure way of checking for SQL injection, it is computationally very intense because it involves searching of the linked list for a matching structure. For a database where the number of valid query structures is very huge, this could take an unacceptably long time. Hence, the technique proposed to deal with this shortcoming is to use a multithreaded search, where the input query is checked with each different query structure running as different threads. When a thread performs a successful search, it intimates all other running threads to immediately terminate.

- However, due to hardware constraints, there is always a limit on the number of simultaneously executing threads in any application. Hence, to counter this, a technique is used where we check for the matching structure with stored

structures based on a priority search, where the queries that are used more regularly are given a higher priority compared to those used rarely, which can be easily maintained by associating a hit counter with each query structure which is incremented each time the particular structure is matched with an incoming query.

- For runtime matching, if a conventional literal matching is used to compare tokens, it will lead to a huge computational complexity. For example, if there are ‘n’ literals in the incoming query and ‘q’ individual valid query structures of the Data related to the tokens of a valid Is it a user input? Link to next node query same length as the input query, the worst case complexity will be O(n*q). To avoid this overhead, a technique is used where instead of using literal string matching algorithms; each token is simply mapped to an integer value. These integer values are also stored in the database instead of the literals as the query fingerprint. When an input query arrives for checking, each token of that query is replaced by its corresponding integer value, and then performing direct integer comparison checks instead of token matching, thus greatly reducing the computational overhead and also significantly reducing the memory space required.

  The formula used to convert a token into its corresponding integer value is to multiply each ASCII decimal value of a literal by its position number in the token, and then sum it up. For example, let’s consider the keyword, ‘SELECT’, the corresponding ASCII decimal values are S=83, E=69, L=76, E=69, c=67, T=84; and the position of each literal is S=1, E=2, L=3, E=4, C=5, T=6. So, after multiplying the ASCII values of each literal with its position and summing them up, we get, 83*1 + 69*2 + 76*3 + 69*4 + 67*5 + 84*6 = 1564. Hence, the corresponding integer value of ‘SELECT’ is 1564. The integer equivalents of all other keywords can be similarly evaluated.

- It is known that for a valid incoming query, the number of tokens is the same as the number of tokens in its corresponding query structure in the database. Therefore, to reduce the search space, all the valid structures having the **same length are grouped together**. For an input query, first its length is calculated,

and it is only compared with that group of valid structures which have the same length. To achieve this, an array is used each element of which contains the starting address of a doubly linked list which again contains the starting addresses of all the singly linked lists of the valid structures of a particular length. Hence, this array contains one element for each different possible query length in the application, with the cell number indicating the query length. This substantially reduces the search space and optimizes the searching process.

5.1.1 Proposed architecture

This scheme would be implemented as a different layer in between the application and the database. It would perform as a virtual database to the application, as it would take the queries from the application program, analyse them, and if found safe, then send them to the database and subsequently send the result set back to the application. As this scheme is totally dependent of the token generation, for which it uses the DBMS parser, it would be specific to different databases as different databases use different keyword sets and function names, as well as different syntax. Hence, we see that it is a wise choice to use the database parser to perform the parsing.

5.1.2 Performance

The main advantage of this model is that since it is multithreaded in nature, it can utilize the features of the modern multi-core processors very efficiently. The basic complexity of this algorithm is in three procedures:

1. **Token separation:** This depends entirely on the database involved because it is wholly dependent on the database parser, since most databases have a different keyword set, syntax and function names. Thus, this factor can be taken to be the same for all implementations.

2. **Token to integer conversion:** This is of the order O(n) where ‘n’ is the total number of unique literals in all the queries put together.

3. **Searching:** Worst case is when it is an unsuccessful search, or when the match is found in the last linked list of any group. If the length of the singly linked list is

‘m’ and there are ‘q’ such linked lists, then the search complexity is O(m*q). The best case complexity will be if we found a match in the first structure, in which case the order will be O(m). If we had used a literal wise checking, then the complexity would have been O(n*q), where ‘n’ is the total number of literals in the query and n>>m.

5.1.3 Shortcomings of the technique

- It can only detect injection attacks where the structure of the query is changed.
- This model can only process a standalone SQL query, but does not work for PL/SQL code block.

5.2 Weight-based Symptom Correlation Approach

This technique has been proposed by Massimo Ficco et al in the research paper titled “A Weight-Based Symptom Correlation Approach to SQL Injection Attacks” [31]. In this technique, a number of symptoms of an SQL injection attack are considered which appear in different times, involve different components and produce several alert events. Correlating these symptoms, which are diverse in nature and detected by distributed probes, allows us “to build a unified view of the web service security, as well as simplifies the recognition of intrusive behaviours”. [31]

Correlation process: Correlation is a process that receives as input detected symptoms from many distributed probes. During this process, symptoms are analyzed and merged into compact reports, which describe the security status of the monitored web applications, which is followed by a confidence assessment of the produced reports. As shown in the figure below [31], the steps performed by the considered correlation process are the following:

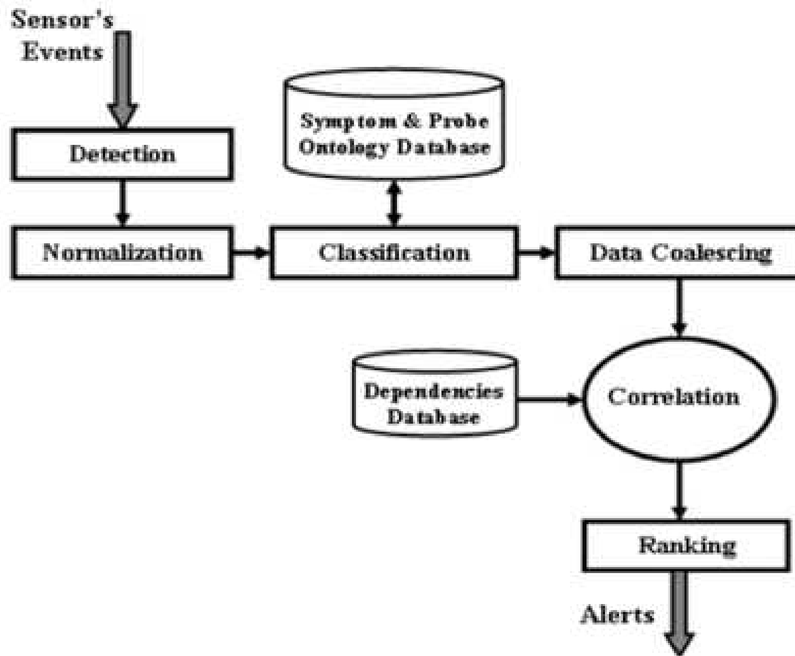


Figure 8: *Correlation Process for weight-based approach*

- **Detection:** Distributed probes and detection mechanisms are used to track different attack symptoms of an SQLIA.
- **Normalization:** Every detected symptom is recorded and normalized into a standardized format, and is also augmented with additional information such as timestamps, source address of attacker, etc.
- **Classification:** Here, the symptoms are aggregated into categories depending on a number of different parameters.
- **Data coalescing:** Events generated by different probes detecting the same symptom are merged into a single event.
- **Correlation:** It receives the classified symptoms and correlates them by using various collaboration rules.
- **Ranking:** Once the correlation succeeds, a decision is taken whether the current observations correspond to malicious activities with respect to the system mission.

In general, the 'system mission' represents the major objectives of the detection process pursued by the security administrator. In this context in particular, the system mission for a SQLIA attack typically consists in avoiding unauthorized access to the backend database and any sensitive information.

Steps involved

1) **Detection:** The use of multiple heterogeneous and distributed probes potentially improves the detection performance through the generation of different perspectives of the same security incident. For example, the length of the query attributes the error code generated by the database server, or the size of the pages returned by the web server could all be used as probes to detect symptoms of an SQLIA. Each probe uses a detection model that allows it to assign a probability value, called an 'anomaly score' (AS), which reflects the probability of the occurrence of the given anomaly with regards to an established profile in keeping with the system mission. Based on this value the evaluated feature is either classified as a potential attack's symptom or as normal.

2) **Normalization:** Since each probe can provide varied security information with differing representations or formats, a process of symptom normalization into a common format is imperative. On the basis of a specific mapping scheme, several attributes are associated with each event to aid the normalization process, such as the identifier of the probe, the symptom identifier, the source and the target of the attack, the start/end times of the symptom, and the anomaly score.

3) **Classification:** Classification aims to categorize symptoms. Categorization schemas must be defined to identify classes of symptoms in a prioritized, hierarchical manner with respect to the overall system mission. Symptoms may be categorized along several dimensions. For example, they could be divided into abuses, misuses, and suspicious acts. Abuses represent actions which change the state of a system's asset, such as sensitive data or database schema, etc. These can further be divided into anomaly based and knowledge-based abuses. The former represent anomalous behaviors (unusual application load, anomalous input

requests); the latter are based on the recognition of signatures of previously known attacks (e:g:, brute force attacks). Misuses represent out-of-policy behaviors in which the state of the components are not affected (e:g:, authentication failed, failure queries). Suspicious acts are not policy violations of any kind but are merely events of interest to the probes (e:g:, commands which provide information about the state of the system).

4) **Data coalescing:** In order to avoid multiple messages referring to the same physical symptom from being generated, events that represent the independent detection (by different probes) of the same symptom occurrence are coalesced to a single event. When two symptoms are merged, the resulting event replaces the constituent events, and will be considered for matching with subsequent events. In particular, the resulting event presents an AS equal to the sum of the ASs of each of them.

5) **Correlation:** In this phase, the different symptom classes are correlated using one of a number of different correlation rules, and a meta-alert consisting of the correlated symptoms is generated. Researchers have proposed several alert correlation techniques and analysis processes. For example, a correlation rule that aggregates symptoms based not only on the impact they have on the system mission but also on their temporal proximity. Impact analysis requires a previous modelling of the relationships between symptoms and mission, which could either be determined through extensive experience and experimentation or through a heuristic-based technique.

6) **Ranking:** In order to reduce the effort required to analyze the volume of generated alerts, an approach based on the confidence of the meta-alerts is adopted. Assuming that S (k) = {fs1; s2; :::; sn} is the set of correlated symptoms during the time window k, the confidence is the probability that the meta-alert represents malicious actions with respect to the system mission.

5.2.1 Detection Approach

In order to detect SQLIA symptoms, anomaly detection models are adopted. They allow to assign a probability value (anomaly score) to the generated events, which reflects the probability of the occurrence of the given anomaly with regards to an established profile. The typical features used to detect symptoms are :

- **Character Distribution (CD):** Typically, SQLIAs present a number of characters that are repeated many times and hence the character distribution can be highly anomalous. Therefore, an anomaly detection model is used to capture the concept of 'normal' query attributes and flag any attempt at SQL injection based on the character distribution. During the training phase, for each HTTP GET and POST request, the query section is extracted, and the relative frequency of each character in the attributes is computed. Then the characteristics of normal character distribution are approximated by the average of all character distributions (the sum of the distributions is divided by the number of requests). The estimated frequencies are sorted and grouped, and any input query differing in its distribution of characters is marked with a corresponding Anomaly Score (AS).

- **Query Length (QL):** The lengths of the inputs given in the different fields of a form that is part of a web request can be used to detect anomalous behaviors. Generally, the lengths of the query attributes do not vary much among requests associated with the same web application. However, this behavior may show considerable deviations during SQLIA's. For example, in UNION attacks, the attacker injects a statement of the form "UNION <injectedquery>", which changes the length of the query attribute quite significantly. A model is adopted which statistically estimates an approximation of the query's attribute length and detects suspicious inputs that significantly deviate from the observed normal behaviour.

- **Queries Failed (QF):** SQLIAs that execute many queries on a particular database table could show up as an anomalous high rate of queries failed with respect to

the normal behaviour. An operational model is adopted to estimate abnormal rate of queries failed with respect to the normal profile. This is considered over a fixed slicing time window.

- **Web Response (WR):** The size of the page generated by the web server when under SQLIA can vary significantly from the size of the corresponding page during normal execution. For example, the web server could generate a web page which contains an error message, whose size is quite different compared to the normal response. In particular, it can be safely assumed that the size of the page generated against the same request does not vary by much and any such anomaly can be flagged as a symptom of an SQLIA. During the training phase it is necessary to estimate the mean and the variance of generated page for each web page directly reachable by the user.

5.2.2 Performance

Weight-based correlation approach for SQLIAs detection allows the system to assign a higher level of confidence to the alerts collected by multiple security probes, located at different architectural levels, so as to achieve a higher probability of spotting an intrusion. In comparison, the other methods are based on a single data source or on multiple data sources, but located at a single architectural level, and are hence not as comprehensive. Weight-based approach is seen to give a very good performance in detecting a majority of both false positives and false negatives.

The injection attacks of the UNION type are very efficiently detected by the Query Length detection while the Tautology attacks are very well detected by the Character Distribution detection. Thus, assigning appropriate weights to these two detection probes could lead to the minimization of false positives. Also, a feedback learning technique could be used whereby the false positives once recorded can be avoided the next time by modifying the weights to generate better anomaly scores.

CHAPTER 6

STORED PROCEDURES

6.1 Introduction

Stored procedures are precompiled database queries that improve the security, efficiency and usability of database client/server applications. Developers specify a stored procedure in terms of input and output variables. They then compile the code on the database platform and make it available to aplication developers for use in other environments, such as web applications. All of the major database platforms, including Oracle, SQL Server and MySQL support stored procedures. The major benefits of this technology are the substantial performance gains from precompiled execution, the reduction of client/server traffic, development efficiency gains from code reuse and abstraction and the security controls inherent in granting users permissions on specific SP instead of the underlying database tables.

A stored procedure has a name, a parameter list, and an SQL statement, which can contain many more SQL statements. There is new syntax for local variables, error handling, loop control, and IF conditions. Here is an example of a statement that creates a stored procedure.

```
CREATE PROCEDURE procedure1              /* name */
(IN parameter1 INTEGER)                  /* parameters */
BEGIN                                    /* start of block */
  DECLARE variable1 CHAR(10);            /* variables */
  IF parameter1 = 17 THEN                /* start of IF */
    SET variable1 = 'birds';             /* assignment */
  ELSE
    SET variable1 = 'beasts';            /* assignment */
  END IF;                                /* end of IF */
  INSERT INTO table1 VALUES (variable1); /* statement */
END                                      /* end of block */
```

Now to Call a Procedure

CALL procedure1(8);

6.2 Why Stored Procedures?

Stored procedures are widely used in all the popular relational commercial database system as it gives the following advantage:

Higher performance: A stored procedure especially composed of several complex queries often runs faster combined than if it had been implemented as, for example, a program running on a client computer which communicates with the database by submitting the SQL queries one by one. As stored procedure is stored in the database server side, by having complex logic run inside the database engine via a stored procedure, numerous context switches and a great deal of network traffic can be eliminated. The database server only needs to send the final results back to the user, doing away with the overhead of communicating potentially large amounts of interim data back and forth [23].

Simplification of data management: Stored procedure allow for business logic to be embedded as an API in the database, which can simplify data management. By providing an API that implements business logic within the database using stored procedures, the need to duplicate logic within client programs is lessened or eliminated. If managed appropriately, this may result in a lesser likelihood of data becoming corrupted through the use of client programs that are out of date, or that have not been updated as intended [23].

Security: The implementation of stored procedure is hidden, and certain logic which need to be secured could be encapsulated by stored procedure and user has no way but to call the stored procedure.

Although SQL stored procedure is core of many business products and online services produced by the company, not enough attention has been paid. It is common misunderstanding that stored procedure won't involve business logic, so the size of the code won't be huge. However, lots of business transactions and database system functions are written as stored procedure. Stored procedure is as important as any other client code written in C or Java.

6.3 SQL Injection in Stored Procedures

Stored procedures, opposite to popular belief, are vulnerable to SQLIA’s [12]. Shown below is an example that illustrates how an attacker can exploit vulnerabilities in a stored procedure, to gain illegitimate access to the system as well as the network resources.

Shown below is a sample stored procedure that accepts ‘uname’ and ‘pword’ as user inputs in a variable length string format.

```
Create procedure p
(IN uname VARCHAR(50),IN pword VARCHAR(50))
BEGIN
        SET @sql = CONCAT('SELECT * FROM uinfo WHERE username=" ',uname,' ”
AND password="',pword,'"');
        PREPARE stmt FROM @sql;
        EXECUTE stmt;
END;
```

(Stored Procedure vulnerable to SQL-Injection)

An interesting observation to be made from the code sample shown above is that there is an EXECUTE system function which allows the user to dynamically build a SQL statement as a string and then execute it. This feature is supported in most business database products. Such dynamically constructed SQL statements provide great user

flexibility. However, they face a great threat from SQLIAs. The process of building an SQL statement could be used by the attacker to change the original intended semantics of the SQL statement.

If the stored procedure ***p*** is called with user inputs ***uname*** and ***pword***, the following query would get executed: **SELECT * FROM uinfo WHERE username='uname' and password='pword'.**

In this scenario, suppose a user gives input for variable ***uname*** as 'anything" OR 1=1 #"' and any string, say "null", for the variable ***pword*** the query would take the form: SELECT * FROM uinfo WHERE username= 'anything" OR 1=1 #"' and password='null'.

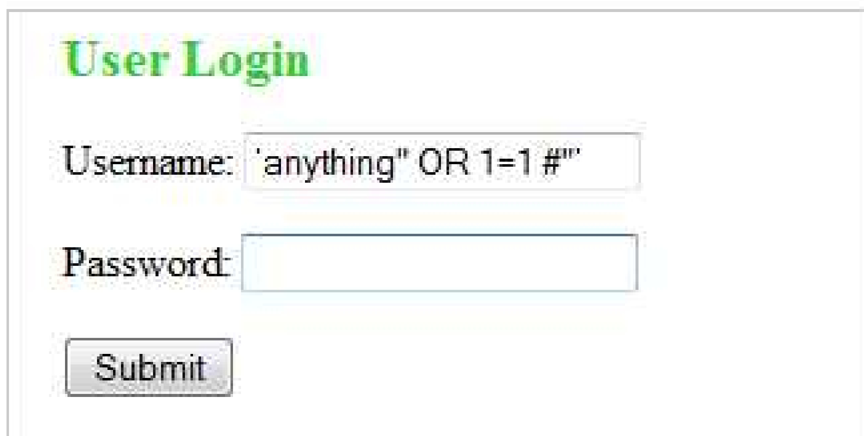


Figure 9: *User Login Form*

Authorization Script In the Web Page:

```
<html >
<head>
<title>SQL Injection Testing</title>
</head>

<body>
<div style="width:300px; height:150px; border:1px #CCCCCC solid; padding:5px;
padding-left:20px;">
```

```
<div style="color: #33CC33; font-size:20px; font-weight:bold;">User Login</div>
<form name="form1" method="post" action="">
 <p>
  <label>Username:</label>
  <input type="text" name="uname">
 </p>
  <p>
  <label>Password:</label>
  <input type="password" name="pword">
 </p>
  <p>
    <label>
    <input type="submit" name="Submit" value="Submit">
    </label>
  </p>
</form>
</div>
<?php
if(isset($_POST['Submit']))
{
	/* Connect to a MySQL server */
	$link = mysqli_connect(
	'localhost', /* The host to connect to */
	'sanu', /* The user to connect as */
	'sanu', /* The password to use */
	'sql_inj'); /* The default database to query */

if (!$link)
{
printf("Can't connect to MySQL Server. Errorcode: %s\n", mysqli_connect_error());
exit();  }
```

```
        /* Send a query to the server */
        if ($result = mysqli_query($link,
        "call p3('".$_POST['uname']."','".$_POST['pword']."')")){

                if(mysqli_num_rows($result))
                {
                        echo "You Authorized User";
                        /* you can now access this application */
                }
                else
                {
                        echo "User Name or Password Failed<br>Please try again";
                }

        }

mysqli_close($link);
}
?>
</body>
</html>
```

The characters ”#” mark the beginning of a comment in SQL, and everything after that is ignored. The query as interpreted by the database is a tautology and hence will always be satisfied, and the database would return information about all users. Thus an attacker can bypass all the authentication modules in place and gain unrestricted access to critical data on the web server.

6.4 Types of Attacks

The basic types of attacks are as follows:

Tautology attacks

The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query's ***WHERE*** conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned. In general, for a tautology-based attack to work, an attacker must consider not only the injectable/vulnerable parameters, but also the coding constructs that evaluate the query results. Typically, the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.

Example:

```
Create procedure p
(IN uname VARCHAR(50),IN pword VARCHAR(50))
BEGIN
	SET @sql = CONCAT('SELECT * FROM uinfo WHERE username=" ',uname,'”
AND password="',pword,'"');
	PREPARE stmt FROM @sql;
	EXECUTE stmt;
END;
```

In this example attack, an attacker submits “ ‘ or 1=1 #

```
SELECT * FROM uinfo WHERE
username=”or 1=1 # AND password=
```

The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them. In our example, the returned set evaluates to a nonnull value, which causes the application to conclude that the user authentication was successful. [4]

UNION attacks

In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query>. Because the attackers completely control the second/injected query, they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

Example: Referring to the running example, an attacker could inject the text "' UNION SELECT cardNo from CreditCards where acctNo=10032 #" into the login field, which produces the following query:

SELECT * FROM uinfo WHERE username='' UNION
SELECT cardNo from CreditCards where
acctNo=10032 # AND password =

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "CreditCards" table. In this case, the database would return column "cardNo" for account "10032." The database takes the results of these two queries, unions them, and returns them to the application.
In many applications, the effect of this operation is that the value for "cardNo" is displayed along with the account information. [4, 21]

Logically incorrect query attacks

This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/injectable parameters to an attacker. Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify injectable parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused the error.

Example: This example attack's goal is to cause a type conversion error that can reveal relevant data. To do this, the attacker injects the following text into input field pin:
"convert(int,(select top 1 name from sysobjects where xtype='u'))".
The resulting query is:
SELECT * FROM uinfo WHERE username='' AND
password= convert (int,(select top 1 name from
sysobjects where xtype='u'))

In the attack string, the injected select query attempts to extract the first user table (xtype='u') from the database's metadata table (assume the application is using Microsoft SQL Server, for which the metadata table is called sysobjects). The query then tries to convert this table name into an integer. Because this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be: "Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int."

There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur. In this case, this value is also the name of the first user-defined table in the database: “CreditCards.” A similar strategy can be used to systematically extract the name and type of each column in the database. Using this information about the schema of the database, an attacker can then create further attacks that target specific pieces of information. [4, 8, 21]

Piggybacked Query

In this attack type, an attacker tries to inject additional queries into the original query. We distinguish this type from others because, in this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that “piggy-back” on the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures, into the additional queries and have them executed along with the original query. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.

Example: If the attacker inputs “’; drop table users #” into the pass field, the application generates the query:

SELECT * FROM uinfo WHERE username=’sanu’ AND
password=’’; drop table users –
After completing the first query, the database would recognize the query delimiter (“;”) and execute the injected second query. The result of executing the second query would be

to drop table users, which would likely destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures.[4, 21, 17]

Buffer overflow

In the buffer overflow attack the user passes more characters to the database, where the number of input characters is limited by the databases. This action can overflow allocated buffer and overwrites adjacent locations in the memory. By crafting the input carefully the attacker can gain access over the database or this type of attack can also confuse the database, thus the database can shutdown unexpectedly. Buffer overflow attacks are normally crafted by making use of security loop holes in the programming. So by installing updated security patches to the databases the administrator can stop these types of attacks. Some of the databases are updated with new technologies can stop any connections coming from the application when a buffer overflow attack is triggered. By making use of this the attacker can trigger the buffer overflow attacks many times. This results the database to stop acting against any commands that are to be executed by the application. This attack can effect normal operation of the application. Bind Variables are another concept that a developer can use to save system resources and to reduce the application execution time. When a command is used against a database the command is saved in shared pool. When a SQL command is passed to database by the application, the database checks in the shared pool to verify whether the command is executed previously or not. If the command is not executed before the database goes through all the process to execute the command, if the database is able to find the result in the shared pool it directly uses the result that is stored in shared pool to response to the database query. So to save the application resources developer‟s uses bind variables with the SQL statements. The attackers can try to manipulate the bind variables to execute applications maliciously. Generally oracle is immune to this type of attacks as oracle will use the value of bind variables exclusively. And the oracle database works such as not to reveal any value from the database, when there are no matching values.

6.5 Preventing SQLIA’s in Stored Procedures

This chapter covers several large areas of secure coding behavior as it relates to SQL injection. First we’ll discuss alternatives to dynamic string building when utilizing SQL in an application. Then we’ll discuss different strategies regarding validation of input received from the user, and potentially from elsewhere. Closely related to input validation is output encoding, which is also an important part of the arsenal of defensive techniques that should consider for deployment.

6.5.1 Validating Input

Input validation is the process of testing input received by the application for compliance against a standard defined within the application. It can be as simple as strictly typing a parameter and as complex as using regular expressions or business logic to validate input. There are two different types of input validation approaches: whitelist validation (sometimes referred to as inclusion or positive validation) and blacklist validation (sometimes known as exclusion or negative validation).

Whitelisting

Whitelist validation is the practice of only accepting input that is known to be good. This can involve validating compliance with the expected type, length or size, numeric range, or other format standards before accepting the input for further processing. For example, validating that an input value is a credit card number may involve validating that the input value contains only numbers, is between 13 and 16 digits long, and passes the business logic check of correctly passing the Luhn formula (the formula for calculating the validity of a number based on the last “check” digit of the card).

When using whitelist validation we should consider the following points:

- ***Data type*** Is the data type correct? If the value is supposed to be numeric, is it numeric? If it is supposed to be a positive number, is it a negative number instead?
- ***Data size*** If the data is a string, is it of the correct length? Is it less than the expected maximum length? If it is a binary blob, is it less than the maximum expected size? If it is numeric, is it of the correct size or accuracy? (For example, if an integer is expected, is the number that is passed too large to be an integer value?)
- ***Data range*** If the data is numeric, is it in the expected numeric range for this type of data?
- ***Data content*** Does the data look like the expected type of data? For example, does it satisfy the expected properties of a ZIP Code if it is supposed to be a ZIP Code? Does it contain only the expected character set for the data type expected? If a name value is submitted, only some punctuation (single quotes and character accents) would normally be expected, and other characters, such as the less than sign (<), would not be expected.

A common method of implementing content validation is to use regular expressions. Following is a simple example of a regular expression for validating a U.S. ZIP Code Contained in a string:

^\d{5}(-\d{4})?$

In this case, the regular expression matches both five-digit and five-digit + four-digit ZIP Codes as follows:

- ^\d{5} Match exactly five numeric digits at the start of the string.
- (–\d{4})? Match the dash character plus exactly four digits either once (present) or not at all (not present).
- $ This would appear at the end of the string. If there is additional content at the end of the string, the regular expression will not match.

In general, whitelist validation is the more powerful of the two input validation approaches. It can, however, be difficult to implement in scenarios where there is complex input, or where the full set of possible inputs cannot be easily determined. Difficult examples may include applications that are localized in languages with large character sets (e.g., Unicode character sets such as the various Chinese and Japanese Character sets).

Blacklisting

Blacklisting is the practice of only rejecting input that is known to be bad. This commonly involves rejecting input that contains content that is specifically known to be malicious by looking through the content for a number of "known bad" characters, strings, or patterns. This approach is generally weaker than whitelist validation because the list of potentially bad characters is extremely large, and as such any list of bad content is likely to be large, slow to run through, incomplete, and difficult to keep up to date.
A common method of implementing a blacklist is also to use regular expressions, with a list of characters or strings to disallow, such as the following example:

```
'|%|--|;|/\*|\\\*|_|\[|@|xp_
```

In general, we should not use blacklisting in isolation, and we should use whitelisting if possible. However, in scenarios where we cannot use whitelisting, blacklisting can still provide a useful partial control. In these scenarios, however, it is recommended that we use blacklisting in conjunction with output encoding to ensure that input passed elsewhere (e.g., to the database) is subject to an additional check to ensure that it is correctly handled to prevent SQL injection.

An example of using preg_match to validate a form parameter:

```
$username = $_POST['username'];
if (!preg_match("/^[a-zA-Z]{8,12}$/D", $username) {
// handle failed validation
}
```

6.5.2 Encoding Output

In addition to validating input received by the application, it is often necessary to also encode what is passed between different modules or parts of the application. In the context of SQL injection, this is applied as requirements to encode, or “quote,” content that is sent to the database to ensure that it is not treated inappropriately. However, this is not the only situation in which encoding may be necessary.

An often-unconsidered situation is encoding information that comes from the database, especially in cases where the data being consumed may not have been strictly validated or sanitized, or may come from a third-party source. In these cases, although not strictly related to SQL injection, it is advisable that we consider implementing a similar encoding approach to prevent other security issues from being presented, such as XSS.

Encoding to the Database

Even in situations where whitelist input validation is used, sometimes content may not
be safe to send to the database, especially if it is to be used in dynamic SQL. For example, a last name such as O’Boyle is valid, and should be allowed through whitelist input validation. This name, however, could cause significant problems in situations where this input is used to dynamically generate an SQL query, such as the following:

$sql = "INSERT INTO names VALUES ('". $_POST[‘fname’]. "','"
$_POST[‘lname’]. "');"

Additionally, malicious input into the first name field, such as:

',''); DROP TABLE names #
could be used to alter the SQL executed to the following:
INSERT INTO names VALUES ('',''); DROP TABLE names--','');

We can prevent this situation through the use of parameterized statements, as covered earlier. However, where it is not possible or desirable to use these, it will be necessary to encode (or quote) the data sent to the database. This approach has a limitation, in that it is necessary to encode values every time they are used in a database query; if one encode is missed, the application may well be vulnerable to SQL injection.
MySQL Server uses the single quote as a terminator for a string literal, so it is necessary to encode the single quote when it is included in strings that will be included within dynamic SQL. In MySQL, can do this by quoting the single quote with a backslash (\).

This will cause the single quote to be treated as a part of the string literal, and not as a string terminator, effectively preventing a malicious user from being able to
exploit SQL injection on that particular query.

Additionally, PHP provides the mysql_real_escape( ) function, which will automatically quote the single quote with a backslash.
mysql_real_escape_string($user);
For example, the preceding code would cause the string O'Really to be quoted to the string O\'Really. If stored to the database, it will be stored as O'Really but will not cause string termination issues while being manipulated while quoted.

CHAPTER 7

ANALYSIS IMPLEMENTATION AND RESULTS

7.1 Implementation

To test the performance of the prevention techniques described in the previous chapter, a system has been developed to detect known form of SQL inject. It is developed in PHP.
We know that SELECT query used in login page of the web application is of the form :
" SELECT * FROM uinfo WHERE username='".$_POST['uname']."' AND password='".$_POST['pword']."' "

In this scenario attacker can easily inject a malicious code ' or 1=1# in the username textbox. The query form by this code is always true because it is a tautology hence anybody can login using malicious code.

To prevent this SQL-Injection, the user input should be validated first, before sending it to database for query processing. If the query is matches to our desire query, we assume that there is no attack in user submitted data otherwise there will be SQL-Injection attack. This idea is illustrated in the following flow chart.

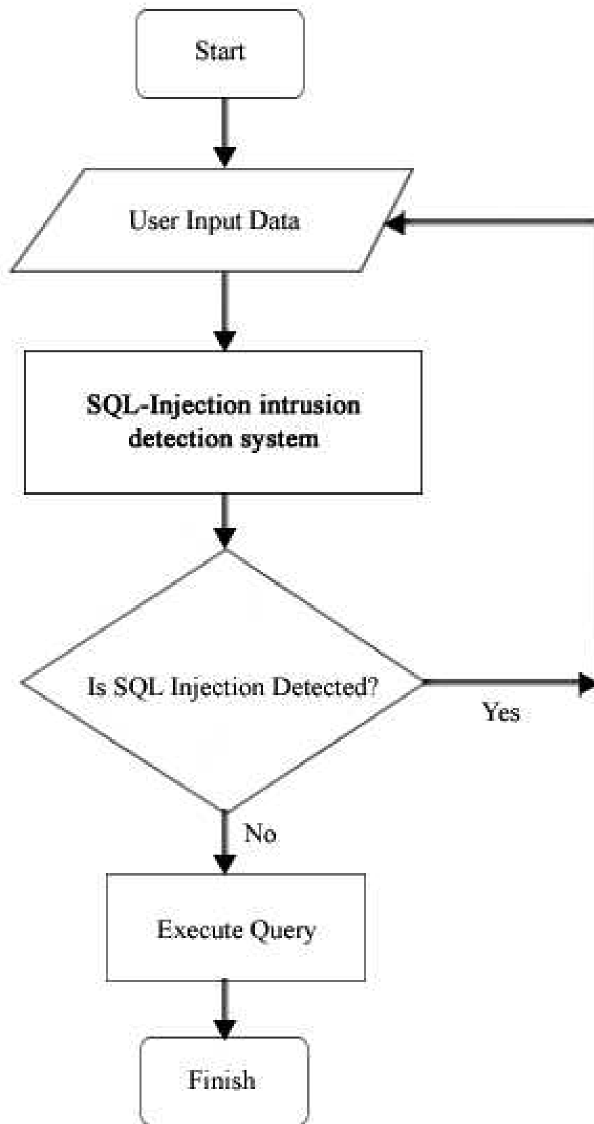


Figure 10: *Flow chart for intrusion detection for SQL-injection*

7.2 Results

Developed SQL-Injection intrusion detection system tested against various stored procedures. The stored procedures were built to work on the standard databases. The database was the **shopping_cart** , with the tables 'uinfo' used for testing the intrusion detection for SQL-Injection. A sample stored procedures used for testing the software is as shown below. This procedure contains 1 query statements with dynamic SQL constructs, and depends on 2 user input parameters.

```
Create procedure p
(IN uname VARCHAR(50),IN pword VARCHAR(50))
BEGIN
        SET @sql = CONCAT('SELECT * FROM uinfo WHERE username=" ',uname,' ”
AND password="',pword,'"');
        PREPARE stmt FROM @sql;
        EXECUTE stmt;
END;
```

The following table gives the summary of the testing results.

Type of injection	Successfully blocked
Tautology	Yes
Union	Yes
Piggy backed	Yes
Logically incorrect query	Yes
End of line comment	Yes

Table 2: *Summary of testing results*

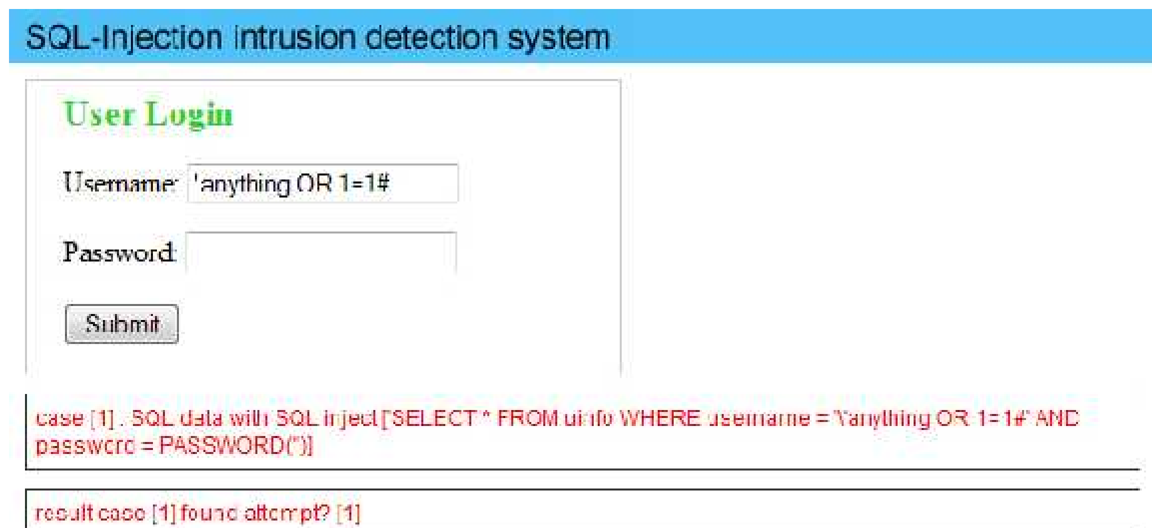


Figure 11: *SQL-Injection intrusion detection*

7.3 Performance Analysis

Number of input versus execution time

The performance evaluation for this case was done by varying the number of input to validate. The execution times for the SQL-Injection intrusion detection (ID) were measured by varying the numbers of user input and the average times were calculated.

The graph shown below gives the relation between the numbers of user input and the average time SQL Injection intrusion detection takes to analyze the inputs. The graph is seen to show an increasing trend, as predicted, where higher the number of user inputs, higher is the execution time.

The average execution time for analysis of a user submitted data having 12 input is 235 ms, which is a reasonable compromise in most cases considering the enhanced level of security it provides.

Sr. No.	**No. of User Input**	**Avg. time in ms**
1	1	20
2	3	58
3	6	118
4	9	179
5	12	235

Table 3: *Execution time for different no. of input*

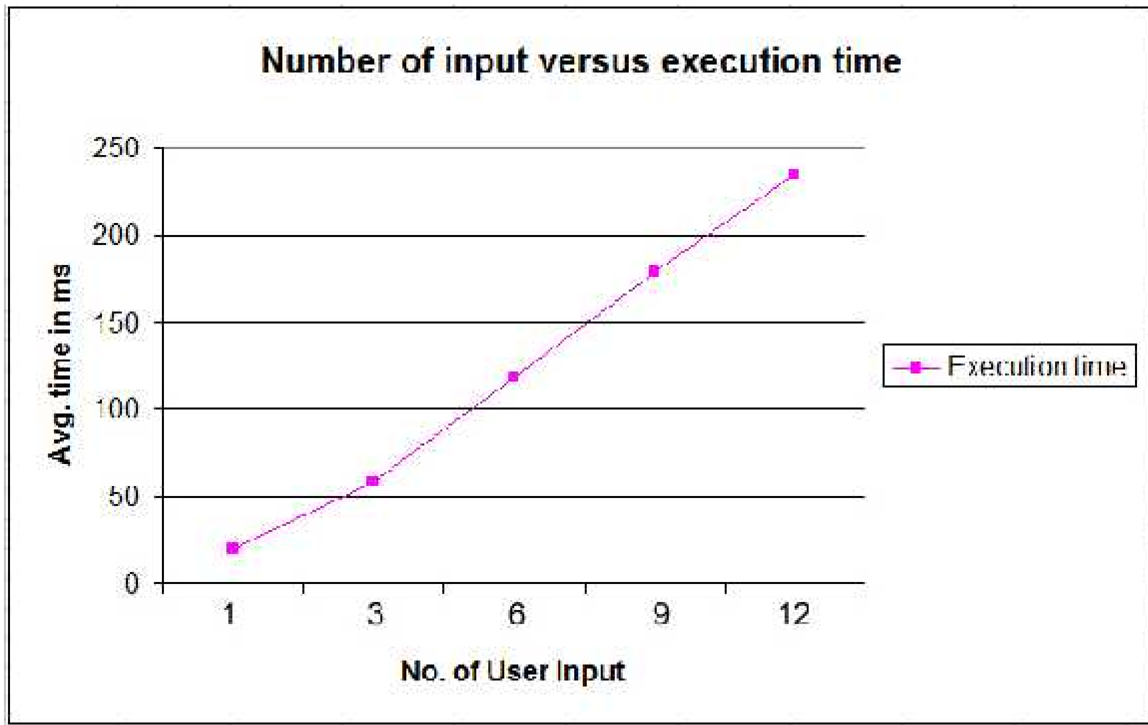


Figure 12: Number of input versus execution time

Number of users versus execution time

The performance evaluation for this case was done by varying the number of users and 1 user input data. The execution times of the SQL-Injection ID were measured by varying the numbers of users and the average times were calculated. Here, we take the response time for 5 and 10 users in 10 times and calculated their average time by taking mean for each of the user. Instead of allowing different machine for different users, all the users processes were simulated on a single machine.

The graph below shows the relation between the number of users and the execution time of the SQL-Injection intrusion detection. As predicted, this graph also shows an increasing trend; higher the users, higher is the execution time.

The average execution time for analysis of a user submitted data having 1 input and 10 users is 160 ms, which is also reasonable compromise in most cases considering security it provides.

Numbers of users	Avg. Response time(ms)
5	34
10	160

Table 4: *Execution time for different no. of users*

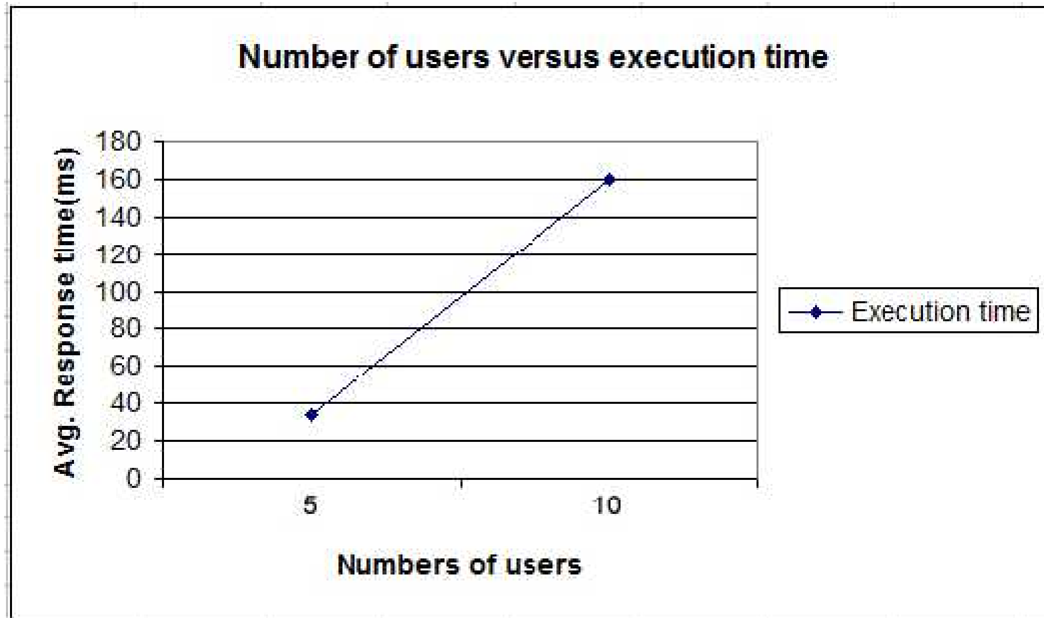


Figure 13: *Number of users versus execution time*

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Concluding Remarks

SQL Injection is a common technique that hackers employ to exploit underlying databases in several web and e-Commerce applications in the present day. These attacks reshape the SQL query, thus altering the behavior of the program. Although several solutions exist to prevent SQLIA's at the application level, very few solutions exists to prevent them from occurring in stored procedures. The main advantage of the model proposed by Dibyendu Aich, an M-tech [30] is that since it is multithreaded in nature, it can utilize the features of the modern multi-core processors very efficiently and in the technique proposed by Massimo Ficco et al [31] the injection attacks of the UNION type are very efficiently detected by the Query Length detection while the Tautology attacks are very well detected by the Character Distribution detection.

In this thesis work, we show the different technique of attacking the Database Stored Procedure through SQL Injection attack and some of their prevention technique. Here we also show the execution overhead for normal statement and guarded statement, which show that more time is need to execute the query in guarded statement than for the normal statement. We can also say that if we follow the prevention technique given in this work our application is secure in some aspect. During the time of this work when we studied various research paper and book we conclude that the main reason of SQL Injection attack is due to the lack of adequate user input validation.

Through the study of different research paper we conclude that security of web application is depends upon the programmer who designs the application. At last we say that during the design of application, user need to give least privilege as possible.

8.2 Future Work

In this thesis work, we concentrated on specific attacking technique on database SP i.e SQL Injection attack. We strongly believe that only this work is not sufficient to secure the web application thus in this field further investigation is needed. In this work we only concerned with some technique of how SQL Injection attack occur in database SP their prevention technique and execution time. During preparation of this work we studied that there are numbers of other factors in the SQL Injection that are harmful to web application but these are not included in this small work.

REFERENCES

[1] Andrew Jaquith, The security of applications: Not all are created equal. Technical report, @stake, feb 2002.
http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf.

[2] Berners-Lee, The World Wide Web browser, http://www.w3.org/People/Berners-Lee/WorldWideWeb.html (1990)

[3] Breach.com, The Web Hacking Incidents Database Annual Report 2009. Breach Security Whitepaper, Feb 2009.
http://www.breach.com/resources/whitepapers/downloads/WP_WebHackingIncidents_2008.pdf

[4] C. Anley, Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.

[5] Chip Andrews, Sql injection faq. Web advisory, apr 2003.
http://www.sqlsecurity.com/DesktopDefault.aspx?tabindex=2&tabid=3.

[6] Craig Atkins, Data sanitization - reducing security holes in an asp website. Web advisory, 2003. http://www.4guysfromrolla.com/webtech/112702-1.shtml.

[7] Dancho Danchev, Over 1.5 million pages affected by the recent SQL injection attacks, ZDNet.com,20th May 2008.
http://blogs.zdnet.com/security/?p=1150

[8] D. Litchfield, Web Application Disassembly with ODBC Error Messages. Technical document, @Stake, Inc., 2002. http://www.nextgenss.com/papers/webappdis.doc.

[9] Garrett, J. J, Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/ideas/essays/archives/000385.php (2005)

[10] Himanshu Khatri, Sql server stored procedures 101. Web advisory, jun 2002. http://www.devarticles.com/printpage.php?articleId=142.

[11] Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D. T., Kuo S.Y, Securing Web application code by static analysis and runtime protection. Proceedings of the 13th International World Wide Web Conference (2004)

[12] Justin Clarke,Rodrigo Marcos Alvarez,Gary O'Leary-Steele, SQL Injection Attacks and Defense,2009.

[13] Livshits, V. B., Lam, M. S., Finding Security Vulnerabilities in Java Applications with Static Analysis. The 14th USENIX Security Symposium (2005)

[14] Kevin Spett, Sql injection - are your web applications vulnerable? Technical report, SPI Dynamics, 2002. http://injection.rulezz.ru/SQLInjectionWhitePaper.pdf.

[15] Martin Eizner, Direct sql command injection. Technical report, The Open Web Application Security Project, 2001. http://qb0x.net/papers/MalformedSQL/sqlinjection.html.

[16] Matthew Levine, The importance of application security. Technical report, @stake, jan 2003. http://www.atstake.com/research/reports/acrobat/atstake_application_security.pdf.

[17] M. Howard and D. LeBlanc, Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.

[18] Mitchell Harper, Sql injection attacks - are you safe? Technical report, Dev Articles, may 2002. http://www.devarticles.com/content.php?articleId=138&page=2.

[19] Ofer Sheza, The Web Hacking Incidents Database Annual Report 2007. Breach Security Whitepaper, 2007
http://www.breach.com/assets/files/resources/breach_security_labs/2008/02/The%20Web%20Hacking%20Incidents%20Database%20Annual%20Report%202007.pdf

[20] OWASP, WebScarab Project, http://www.owasp.org/

[21] S. McDonald, SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002.

[22] Steve Christey, Vulnerability Type Distributions in CVE. cwe.mitre.org, Oct 2006.
http://cwe.mitre.org/documents/vuln-trends/index.html

[23] Stored procedure. http://www.wikipedia.org (retrieval date: May 18, 2006), 2005.

[24] The Open Web Application Security Project. A guide to building secure
web applications, Version 1.1.1. Online Documentation, sep 2002.
http://www.owasp.org/.

[25] Thomas Connolly, Carolyn Begg, and Ann Strachan, Database Systems - A Practical Approach to Design, Implementation, and Management. Addison - Wesley, 1999.

[26] W. Halfond, J. Viegas, and A. Orso, A Classification of SQL-Injection Attacks and Countermeasures. Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE), 2006

[29] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, A classifcation of SQL injection attacks and countermeasures. In IEEE International Symposium on Secure Software Engineering, 2006.

[30] Dibyendu Aich (NIT Rourkela). ‘Secure Query processing by blocking sql injection’. M.Tech thesis 2009

[31] Massimo Ficco, Luigi Coppolino and Luigi Romano. ‘A Weight-Based Symptom Correlation Approach to SQL Injection Attacks’.Dependable computing 2009.